Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Integración de un sistema de reglas en un sistema SCADA distribuido

Estudante: Jorge Sardina Pereira
Dirección: Laura M. Castro Souto

A Coruña, setembro de 2020.

A mi familia

Resumen

El objetivo principal del proyecto es la integración de un sistema inteligente basado en reglas dentro de un sistema SCADA escrito en Erlang. El sistema se utiliza para la monitorización y control de sistemas de climatización. Intentamos dotar al sistema de la capacidad de actuar con la mínima supervisión para las tareas de mantenimiento o emergencias. Para ello se diseña un nuevo subsistema que acomode el motor de reglas, se realiza la implementación en Erlang y se hacen las pruebas correspondientes. Posteriormente se extiende la interfaz web para permitir la gestión del nuevo subsistema.

Abstract

The main objective of this project is the integration of an intelligent rules based system into an existing SCADA system written in Erlang. The system is used for the monitoring and control of air conditioning systems. We try to add the capabilities to the system to act with a minimum supervision for the tasks of maintenance or emergencies. For that, a new subsystem is designed for the rules engine. The implementation is done in Erlang with the appropiate tests. After that the web interface is extended to allow the handling of the new subsystem.

Palabras clave:

- Erlang
- Sistema de reglas
- Programación funcional
- Sistema distribuido

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El proyecto que en esta memoria se expone lleva por título **"Integración de un sistema de reglas en un sistema SCADA distribuido"**. A continuación se expone la motivación por la cual se realiza este proyecto y se plantean los objetivos a alcanzar.

1.1 Motivación

ETHIC es un sistema SCADA (Supervisory Control And Data Acquisition) desarrollado para la empresa Arce Clima S.L. con el fin de realizar la supervisión y el control de instalaciones de climatización, así como de sus componentes. Dicho sistema cuenta con una interfaz web desde la que los operarios pueden acceder a todas las instalaciones que controla. Estas instalaciones pueden contar además con múltiples zonas independientes (por ejemplo, en una tienda de ropa se realiza una gestión diferente de la zona de almacenes que de la zona de ventas), cada una con configuraciones diferentes.

En su interfaz web, como se puede apreciar en las figuras 1.1 y 1.2, se muestran los datos de una forma sencilla y estructurada, con gráficas y controles para regular los parámetros. Con esta interfaz los técnicos pueden realizar modificaciones en las instalaciones remotamente, pero el sistema no puede reaccionar ante cambios imprevistos por si mismo, ni alertar de funcionamientos inadecuados. Cada instalación cuenta con un gran número de sensores que generan una gran cantidad de datos que deben comprobarse y verificarse manualmente, tarea que se complica cuantas más instalaciones se añaden.

Así, el sistema necesita evolucionar para permitir realizar un control sobre todos estas instalaciones sin supervisión, facilitando el trabajo de los técnicos de manera que puedan automatizar las comprobaciones y revisiones del sistema.

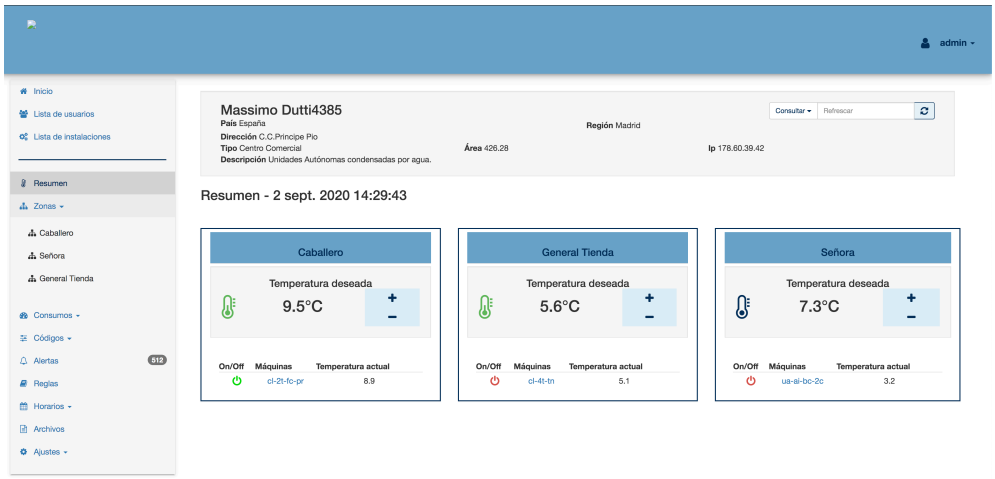

Figura 1.1: Interfaz ETHIC

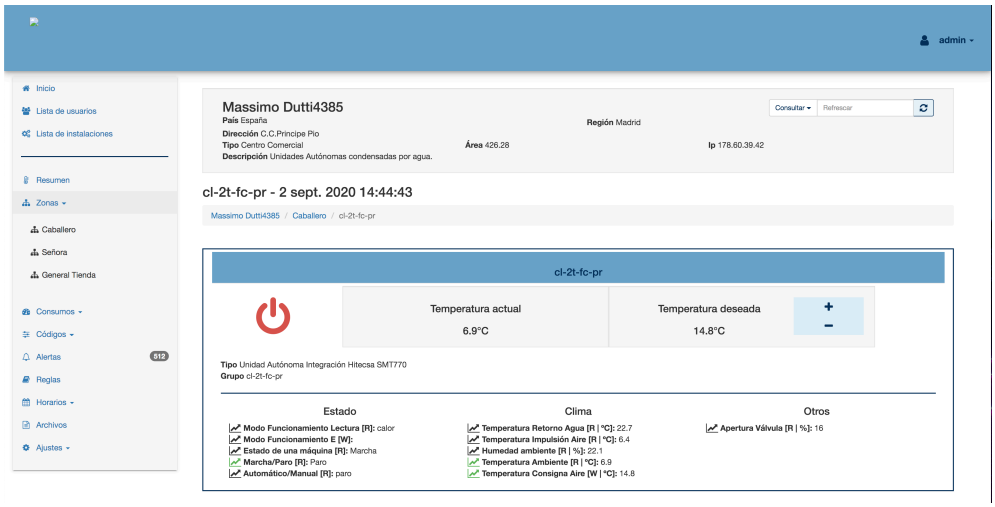

Figura 1.2: Interfaz zona concreta

1.2 Objetivos

Este proyecto busca, en primer lugar, implementar e integrar en el sistema existente ETHiC un motor de reglas que permita gestionar y actuar rápidamente ante la llegada de nuevos datos, con mínima supervisión por parte de los técnicos. Y en segundo lugar, extender la interfaz existente para poder gestionar este sistema de reglas. El sistema tendrá acceso a todos los datos almacenados en la base de datos, y podrá utilizarlos para decidir la ejecución de sus reglas. Los usuarios de la aplicación ETHIC son técnicos con conocimiento sobre el dominio, que gestionarán las reglas para el sistema, pudiendo crear, eliminar o modificar reglas existentes.

Un motor de reglas es un sistema experto formado por un motor de inferencia y una base de conocimiento que contiene las reglas. El motor de inferencia es el cerebro del sistema experto. Este maneja todas las reglas y hechos contenidos en la base de conocimiento. Su trabajo es coger las reglas, aplicarlas a los datos entrantes y ejecutar las acciones correspondientes. Las reglas, en general con un formato *if-then*, contienen una serie de condiciones y una serie de acciones que se ejecutarán en caso de cumplirse las condiciones.

ETHIC está desarrollado con Erlang/OTP del lado del servidor (backend) y Typescript con el framework Angular en la parte de interfaz (frontend). Erlang, por ser un lenguaje funcional, resulta idóneo para el desarrollo de un motor de inferencia. El *pattern matching* facilita la programación de un sistema elegante y sencillo, fácil de extender con nuevas funcionalidades. Por otro lado, Angular es un framework web probado y testeado para hacer interfaces web.

Capítulo 2

Contextualización

En esta parte de la memoria se realiza una contextualización del sistema, su origen y se muestran las herramientas y tecnologías utilizadas.

2.1 ETHiC

ETHiC es un sistema SCADA [1] desarrollado para la empresa Arce Clima S.L.. Arce Clima es una empresa gallega centrada en la instalación industrial de grandes equipos de climatización. Además de realizar instalaciones, Arce Clima también gestiona tareas de asesoría, ingeniería y mantenimiento. Es aquí donde surge ETHiC, como una colaboración entre un equipo de la UDC y Arce Clima para desarrollar un sistema SCADA para gestionar el mantenimiento y buen funcionamiento de todas las instalaciones. Los sistemas SCADA (Supervisory Control and Data Acquisition) son sistemas de automatización y control que llevan años utilizándose para comunicarse con dispositivos de campo en el proceso industrial. Estos han ido evolucionando, desde los primeros sistemas que eran poco más que terminales con datos de telemetría de las instalaciones, a los actuales sistemas que gestionan y controlan una gran cantidad de dispositivos de forma concurrente. Los sistemas SCADA facilitan a los operarios las tareas de supervisión ya que les permiten acceder de manera remota a los datos. El trabajo de mantenimiento se vuelve más ágil gracias a estos sistemas.

2.2 Sistema de Reglas

Los sistemas basados en reglas [2, 3], o sistemas expertos basados en reglas, son la forma más simple de inteligencia artificial. Utilizan reglas como la representación del conocimiento dentro del sistema. Son sistemas basados en el conocimiento, que simulan el razonamiento que haría un experto al resolver un problema. Un sistema de reglas, al ser expuesto a los mismos datos, debería actuar de manera similar a un experto.

Las reglas se expresan como un conjunto de declaraciones if-then, o reglas de producción (`IF P THEN Q`).

Un sistema de reglas consiste en 3 elementos básicos:

- Una serie de *hechos* relevantes para el estado del sistema.
- Una serie de *reglas*. Contienen las acciones que deben tomarse ante un problema. Un sistema debe contener las reglas mínimas y evitar reglas irrelevantes para mantener un buen rendimiento.
- Una condición para terminar. Esto determina si se ha encontrado una solución, o si no existe.

Se pueden clasificar los sistemas de reglas en función del tipo de algoritmo de inferencia que apliquen para alcanzar la solución al problema.

- Encadenamiento hacia adelante. Es un método de razonamiento dirigido por los datos, es decir, en función de los datos que llegan, alcanza una conclusión final. Esta sería la forma habitual de razonamiento de las personas.
- Encadenamiento hacia atrás. En este caso se trata de un razonamiento dirigido por el objetivo. En este caso solo se utilizan los hechos relevantes, para obtener el objetivo buscado.
- Un sistema que aplique ambos algoritmos.

Los sistemas de reglas tuvieron gran popularidad en los años 70, y fueron gran cantidad de los sistemas expertos desarrollados. Pero su principal problema, es que cuanto más crece la cantidad de reglas, más interacciones indeseables entre ellas se producen, por lo que se tienen que aplicar cambios a las reglas existentes.

2.3 Erlang/OTP

Erlang/OTP [4] es un lenguaje de programación funcional desarrollado originalmente en el laboratorio de ciencias de la computación de Ericsson. OTP (Open Telecom Platform) es un conjunto de librerías escritas en Erlang. Sus principales características son:

- **Concurrencia**: La mayoría de las primitivas proporcionadas por Erlang proporcionan soluciones para desarrollar grandes sistemas altamente concurrentes. Su modelo está basado en procesos ligeros y paso de mensajes asíncronos.

- **Sistemas distribuidos**: El paso de mensajes permite la construcción sencilla de sistemas altamente distribuidos, pudiendo pasar de aplicaciones en un único ordenador a una red de ordenadores.

- **Noción del tiempo:** el sistema de Erlang tiene integrada la noción del tiempo, es decir, el programador puede especificar facilmente cuanto debe un proceso esperar un mensaje para tomar una acción. Esto permite la programación de aplicaciones en tiempo real.

- **Tolerancia a fallos:** Erlang ha sido diseñado con varios mecanismos para la detección de errores y gestión de estos durante la ejecución.

- **OTP:** [5] Sin ser exactamente una parte del lenguaje, los principios de diseño de OTP definen como estructurar un programa Erlang. Los conceptos básicos, como el árbol de supervisión de procesos donde un proceso (el supervisor) se encarga de supervisar al resto de procesos (trabajadores) son los pilares de gran parte de los programas en Erlang. OTP define a su vez patrones, *Behaviours*, para los procesos trabajadores. Siendo la idea dividir el código en una parte genérica (el módulo del *behaviour*) y la propia lógica específica de este.

2.4 Angular

ETHiC utiliza una aplicación Angular como interfaz web. Angular [6] es un framework desarrollado por Google para el desarrollo de aplicaciones web utilizando HTML y Typescript. La arquitectura (2.1) de las aplicaciones en Angular se basa en una serie de conceptos:

- NgModules: Los bloques básicos de las aplicaciones en Angular. Los *NgModules* definen un contexto común para los componentes. Permiten organizar el código en distintos módulos funcionales para diseñar aplicaciones complejas pensando en la reusabilidad.

- Componentes: Cada componente define una clase con sus datos y lógica asociados y un template HTML que define la vista a ser mostrada. Un template enlaza los datos de la aplicación y la vista del navegador.

- Servicios: La lógica que no está asociada a una vista y que se va a compartir entre componentes reside en los servicios. Angular permite acceder a los servicios dentro de los componentes utilizando inyección de dependencias.

- Inyección de dependencias: La inyección de dependencias es un patrón en el que una clase obtiene sus dependencias de una fuente externa en vez de crearlas. Angular tiene

su propio framework de inyección de dependencias, este se encarga de proveer a la clase de las dependencias solicitadas al ser instanciada.

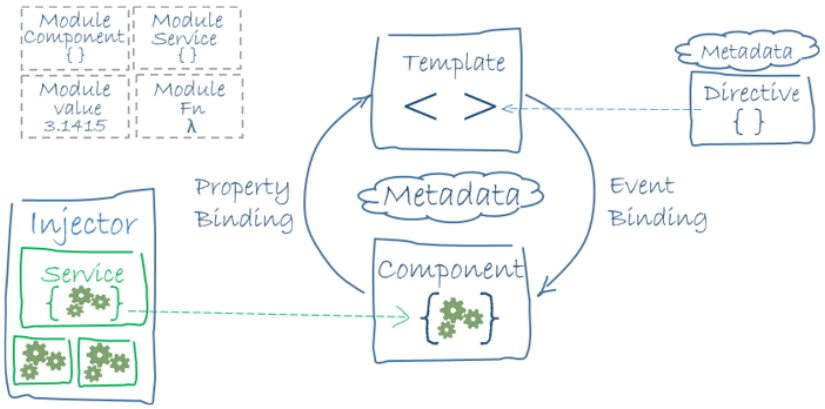

Figura 2.1: Arquitectura de una aplicación en Angular tomada de angular.io

2.5 MongoDB

MongoDB es una base de datos multiplataforma orientada a documentos que forma parte de la familia de bases de datos no relacionales NoSQL. MongoDB almacena los documentos en un formato similar a JSON, BSON. A diferencia de otras bases de datos, en MongoDB no es necesario seguir un esquema, por lo que los documentos de una misma colección no tienen porque tener los mismos campos.

MongoDB ha sido diseñada desde el principio para aplicaciones que puedan requerir escalar horizontalmente, e incluye funcionalidades para ello en la forma de opciones de replicación y distribución con *sharding*.

MongoDB incluye a su vez sistemas nativos de validación de documentos, herramientas de automatización para el respaldo de la base de datos y su propio sistema de búsqueda, ya que las búsquedas no se realizan con el tradicional SQL.

ETHiC utiliza MongoDB para almacenar los datos de todo el sistema, lo cual condiciona como gestionar y almacenar las reglas.

2.6 Git

Git [7] es una herramienta de control de versiones diseñada por Linus Torvalds en 2005 para el desarrollo del kernel Linux. Git forma parte de la familia de los DVCSs (Distributed Version Control Systems), en los cuales los clientes no descargan simplemente la última versión

de los archivos, sino que descargan el repositorio entero, con todo su historial. Cada cliente posee su propio repositorio local y la mayoría de las operaciones en Git, se realizan sobre ficheros locales, sin necesidad de contactar con el servidor. Esto permite que cada desarrollador pueda trabajar independientemente, incluso sin conexión a internet.

En Git, la mayoría de las operaciones simplemente añaden datos al sistema, lo cual hace que sea difícil hacer algo que no se pueda deshacer. Esto hace que usar Git sea muy cómodo para muchos desarrolladores, y ha ayudado, junto a sus otras cualidades, a popularizar el sistema.

En este proyecto se utiliza BitBucket de Attlasian como repositorio central para el proyecto.

2.7 Notion

Notion es una herramienta multiplataforma diseñada para organizar el trabajo en un único lugar. Está orientado tanto como a individuos, como para equipos. Permite tomar notas, escribir documentos, gestionar proyectos sin necesidad de utilizar más softwares.

En este proyecto vamos a utilizarlo para aplicar una metodología Kanban. El método Kanban [8] en desarrollo de software busca proveer de un sistema visual para gestionar los proyectos, generalmente utilizando un tablero con varias columnas, donde se añaden las tareas.

Notion permite, además de aplicar el método Kanban utilizando la vista de tableros como se ve en la figura 2.2, mantener todo el conocimiento relacionado con la aplicación en un mismo sistema. Además permite incluir texto en LaTeX o importar de otras aplicaciones, como Trello.

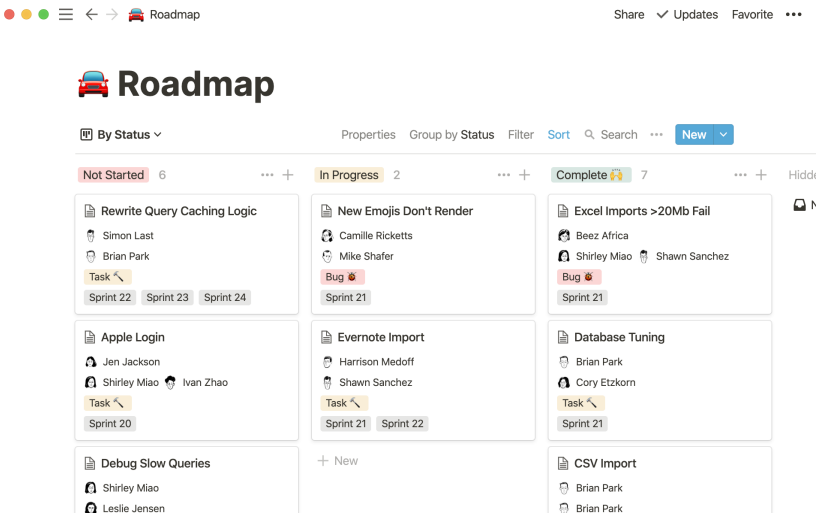

Figura 2.2: Ejemplo de tablero Kanban en Notion

Capítulo 3

Análisis

En esta sección de la memoria se realiza tanto el análisis de los requisitos funcionales y no funcionales como de los casos de uso.

3.1 Análisis de requisitos

Se mantuvieron un par de reuniones con el equipo de Arce Clima para extraer la funcionalidad que debe tener el sistema de reglas. De estas reuniones se obtuvieron una serie de reglas a partir de las cuales se implementó la funcionalidad que debía tener el sistema de reglas para poder ejecutar estas y futuras reglas. Hacia el final del proyecto el contacto con los técnicos de Arce Clima se vio descontinuado, pero este proyecto siguió adelante, por lo que en este punto se decide, para facilitar la tarea de mantener las reglas del motor, añadir una interfaz de control sobre este.

Los requisitos funcionales son los siguientes:

- **Motor de reglas:** El motor de reglas debe de tener la capacidad de actuar sin supervisión. El sistema de reglas debe poder crear alertas en el sistema integrado de alertas de la aplicación. Debe ser capaz de leer todos los campos de la base de datos para poder crear reglas con ellos. Las reglas deben de ser globales para evitar la necesidad de replicarlas por cada instalación.
- **Gestión de reglas:** La aplicación debe permitir crear, eliminar y editar las reglas. Se aplicará el control de usuarios para evitar que usuarios que no sean administradores o técnicos tengan acceso a las reglas globales. Se debe permitir modificar todos los campos de estas.

Requisitos no funcionales:

- **Integración:** La aplicación debe mantener todas las funcionalidades existentes. La integración del sistema de reglas no debe afectar al rendimiento existente, ni a la seguridad

de la aplicación.

- **Rendimiento:** La ejecución de múltiples reglas no debe degradar el funcionamiento del resto del sistema.
- **Usabilidad:** La aplicación web debe ser fácil de utilizar e intuitiva. Debe ser sencilla para los técnicos y administradores del sistema.

3.2 Casos de uso

La aplicación ETHiC incluye un sistema de usuarios, con distintos tipos de usuarios, cada uno con acceso a ciertas funcionalidades. En este caso, al motor de reglas y su interfaz, tendrán acceso el administrador del sistema y el técnico de mantenimiento como usuarios autorizados. Los actores serán:

- Administrador del sistema. Tiene acceso a la totalidad del sistema.
- Técnico de mantenimiento. Tiene acceso a la creación, edición y eliminación de reglas.
- Sistema de reglas. El sistema de reglas puede crear alertas.

Los principales casos de uso del sistema (cf. figura 3.1) son:

- **Creación de reglas:** Este caso de uso permite crear reglas nuevas. Solo los usuarios autorizados pueden crear reglas nuevas.
- **Edición de reglas:** Este caso de uso permite editar reglas existentes. Solo los usuarios autorizados pueden crear reglas nuevas.
- **Activación/Desactivación de reglas:** Este caso de uso permite activar y desactivar reglas. Solo los usuarios autorizados pueden activar o desactivar reglas.
- **Visualizar reglas activas:** En este caso de uso los usuarios podrán ver que reglas están funcionando en el sistema.
- **Crear alertas:** Se podrán crear alertas en el sistema de alertas integrado. El sistema de reglas podrá crear alertas.

3.3 Matriz de trazabilidad

En la tabla 3.1 se expone la correspondencia entre requisitos funcionales y casos de uso.

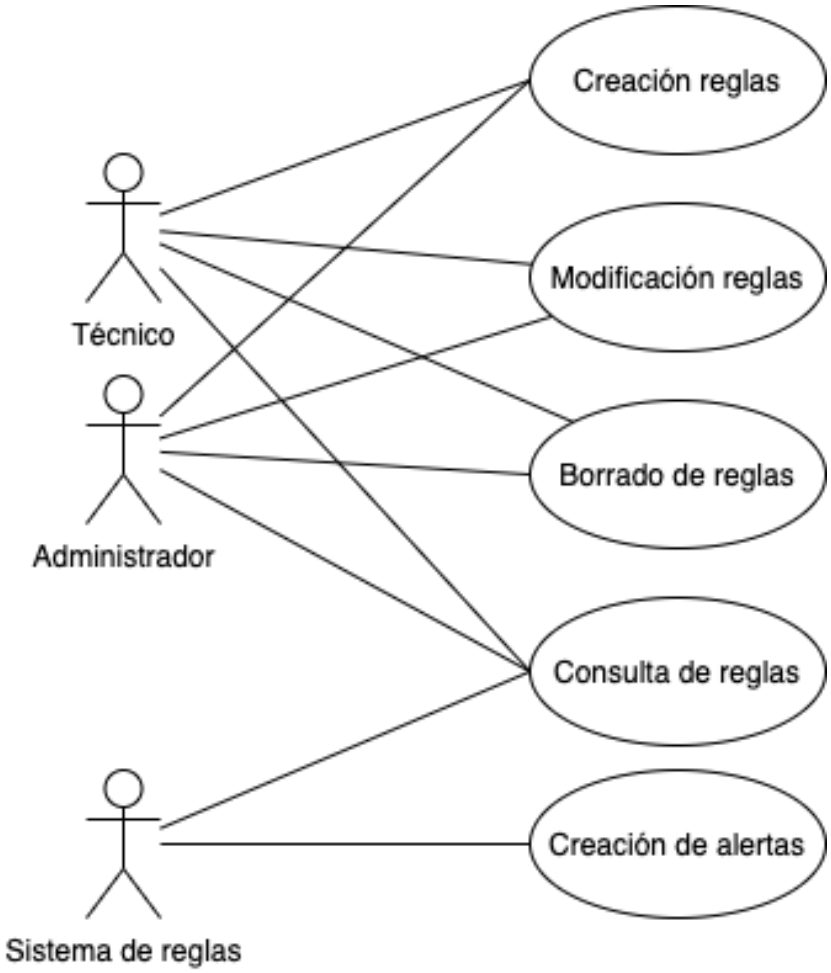

Figura 3.1: Casos de uso

Requisitos / Casos de Uso	Creación de reglas	Modificación de reglas	Borrado de reglas	Consulta de reglas	Creación de alertas
Motor de reglas				x	x
Gestión de reglas	x	x	x	x	

Tabla 3.1: Matriz de trazabilidad de casos de uso

Capítulo 4

Diseño

En esta sección de la memoria se presenta el modelo de datos para el sistema de reglas y la arquitectura que se va a implementar.

4.1 Modelo de datos

Para construir la base de conocimiento debemos añadir una serie de modelos a la aplicación [9]. La aplicación utiliza una base de datos no relacional, por lo que los modelos que utilizaremos serán modelos de datos no relacionales.

Añadiremos tres entidades nuevas al sistema:

- **Reglas**. La entidad principal, las reglas contienen cada una un nombre único, una lista de hechos, una lista de acciones a realizar y un operador que decide cuantos hechos deben cumplirse para ejecutar las acciones, el operador puede ser de tres formas:
  - **All**: cuando deben cumplirse todos los hechos.
  - **Any**: cuando debe cumplirse al menos un hecho.
  - **None**: cuando no debe cumplirse ninguno de los hechos.

  Tambien contiene una marca de borrado.

- **Hechos**. Una regla puede tener varios hechos. Cada hecho contiene un valor a leer de base de datos, el valor a comparar y un operador de comparación. El valor a comparar se obtendrá de base de datos. El operador puede ser de igualdad, desigualdad o comparaciones de cantidad.

- **Acciones**. Las operaciones que se llevarán a cabo en caso de cumplirse la regla. Contienen el tipo de la acción más un campo de parámetros.

Las reglas contienen todas un campo de borrado lógico ya que es el método utilizado en el resto de la aplicación. Este método de borrado suele utilizarse cuando no se quiere eliminar posible información importante.

```
{
    _id: <Object_id>,
    name: <Name>,
    operator: <Operator>,
    facts: [{
        cond: <Condition operator>,
        expected: <Database field>,
        value: <Memory value>
    }...],
    actions: [{
         action: <Action type>,
         options: <Action option>
    }...]

}
```

Modelo de regla

4.2 Arquitectura

La arquitectura de ETHiC sigue un patrón líder-trabajador. En esta patrón, un proceso ”líder”, reparte el trabajo entre los nodos ”trabajadores”. Este tipo de patrón es muy común en Erlang/OTP ya que OTP incluye en sus librerías un modelo de supervisión de procesos.

En el caso de ETHiC (4.2), un supervisor global crea supervisores para cada instalación que gestiona la aplicación, y este a su vez crea trabajadores para cada funcionalidad. Este supervisor, implementa con estos procesos trabajadores, una arquitectura en capas MVC (Model-View-Controller):

- **Vista**: Capa con la interfaz de usuario.
- **Controladores**: ETHiC tiene varios módulos que delegan la entrada de datos al modelo correspondiente.
- **Modelo**: Donde reside la lógica de negocio.
- **Acceso a base de datos**: Capa que realiza las consultas, inserciones y borrado de documentos en la base de datos.

Como esta arquitectura ya está probada, y es adecuada para resolver el problema, se procede a integrar el sistema de reglas manteniéndola. Para cada regla, el supervisor lanza un

proceso trabajador, con el objetivo de que si esta regla tiene un mal funcionamiento, no bloquee todo el sistema ETHiC. Los procesos Erlang son procesos ligeros, y permiten tener una gran cantidad de reglas sin afectar al rendimiento. Siguiendo la arquitectura de la aplicación, se crea un controlador y un modelo para las reglas. El controlador delega en el modelo las llamadas que recibe (4.3). El modelo se encarga de obtener las reglas, crearlas, modificarlas y marcarlas para borrado conectándose directamente al DAO.

En la vista, la aplicación sigue la arquitectura de Angular ya comentada anteriormente (2.1). Se van a añadir dos componentes, uno para la visualización de las reglas y otro para la edición y creación de estas. Cada uno con su correspondiente *template*. Ambos componentes utilizarán un servicio para conectarse con la API del backend.

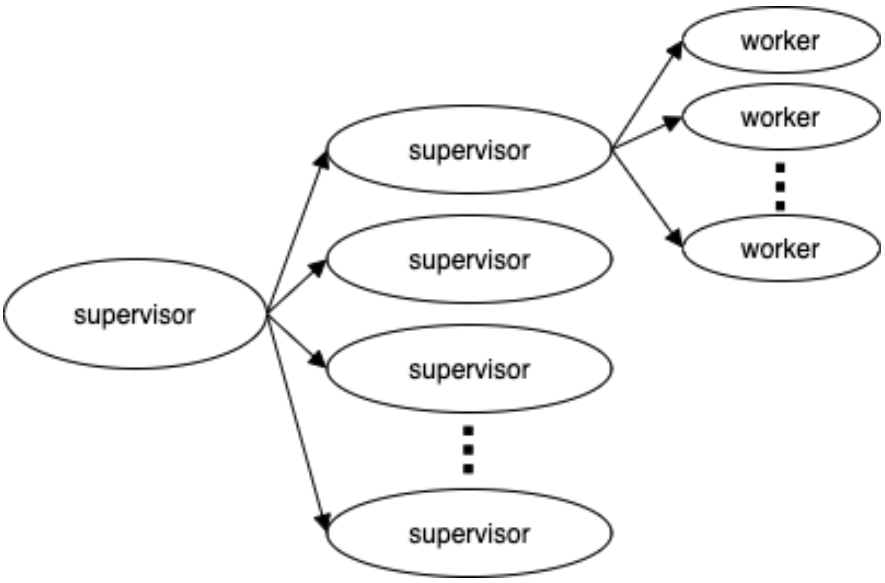

Figura 4.1: Patrón líder-trabajador en ETHiC

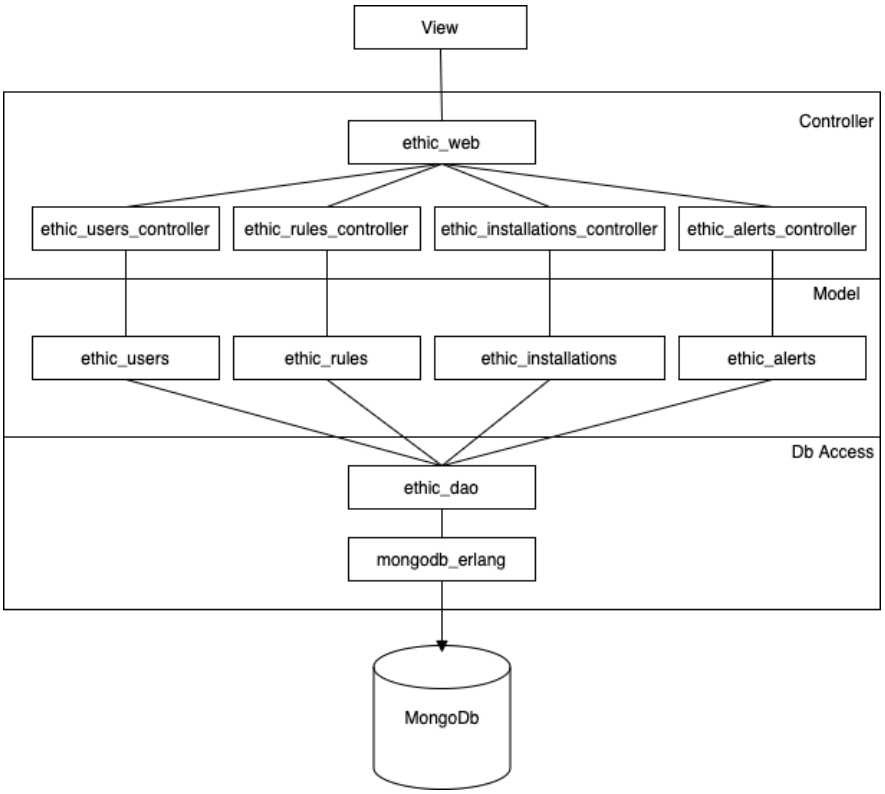

Figura 4.2: Arquitectura MVC en ETHiC

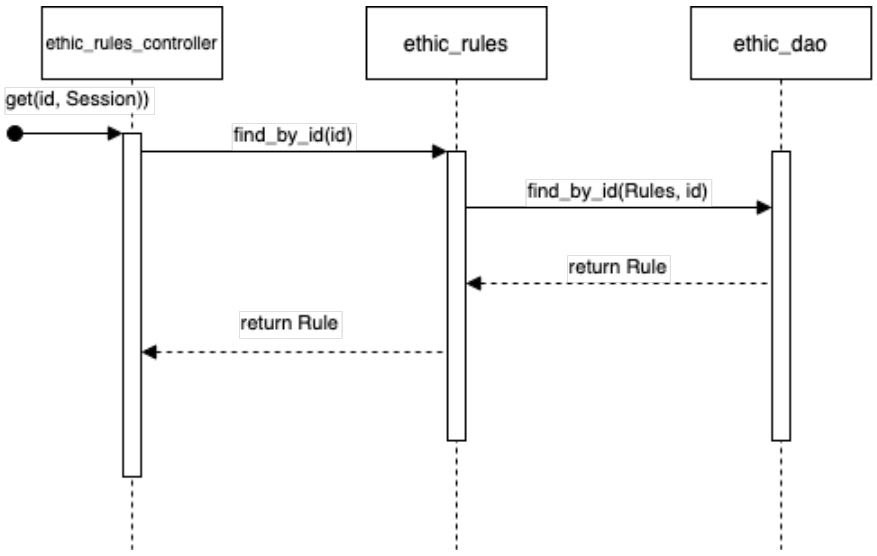

Figura 4.3: Diagrama de secuencia controlador-modelo ETHiC

Capítulo 5

Implementación

En esta sección de la memoria se trata la implementación del sistema, junto a los problemas que surgieron en el desarrollo. La implementación de la aplicación se realizó en dos partes marcadas por una parada en el tiempo del desarrollo, la primera parte centrada en el backend de la aplicación y la segunda en el frontend.

5.1 Backend: Modelo y Sistema de reglas

Al comenzar el proyecto, se buscan ejemplos de proyectos similares en Erlang. Uno de los ejemplos mas relevantes que se encuentra, es Seresye [10]. Seresye permite crear aplicaciones de inteligencia artificial a programadores en Erlang, que escriban sus propias reglas como funcionas Erlang. Esto ya limita su uso a programadores, cuando en esta aplicación intentamos facilitar a usuarios no técnicos incluir reglas en el sistema. Pese a esta diferencia fundamental, se utiliza igualmente el *pattern matching* (ver figura 5.2) de Erlang para decidir la ejecución de las reglas, así como su formato en forma de tupla nativa. (ver figura 5.1). Se experimenta con otros formatos como los Zippers [11], pero se descartan al complicar el código y no mejorar el rendimiento notablemente.

En primer lugar, tras analizar el tipo de datos mayoritariamente numéricos de la aplicación, se crean las condiciones y el código para comprobarlas (ver figura 5.3). Se accede a base de datos y se comprueban los valores especificados en las reglas, devolviendo si se cumple o no.

En este punto del desarrollo, tras reuniones con el equipo de ArceClima, es cuando se decide que las reglas podrán crear alertas, por lo que se crea el primer tipo de acción *alert*, para emitir una alerta al ejecutarse y cumplirse la regla.

Tras desarrollar la base del motor de reglas se procede a integrarlo en el proyecto. Aquí se adopta la arquitectura que ya seguía la aplicación, se crean tanto un controlador para las reglas, como un modelo en el que se delega la funcionalidad de creación, edición, borrado y

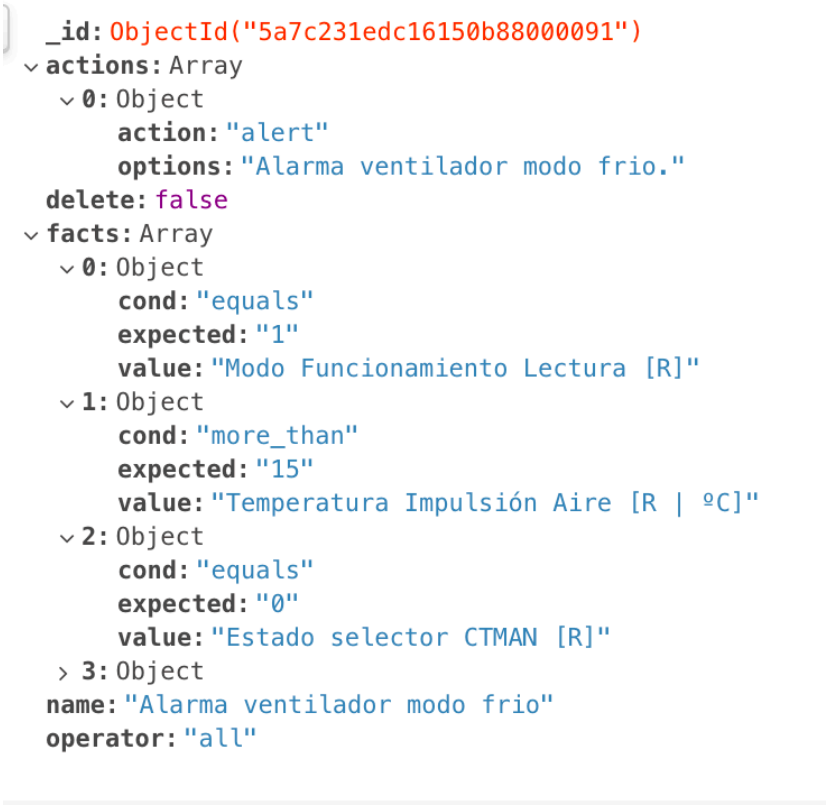

Figura 5.1: Regla en base de datos

```
    -spec run_rule(Id::string(), Rule::tuple()) -> ok | no_match.
run_rule(Id, {"all", Facts_list, Actions_list}) ->
...
run_rule(Id, {"any", Facts_list, Actions_list}) ->
...
run_rule(Id, {"none", Facts_list, Actions_list}) ->
```

Figura 5.2: Pattern matching en las reglas

obtención de reglas.

En el momento de implementar el caso de uso de creación de reglas se encuentra el primer problema, la base de datos almacena documentos, y el motor de reglas utiliza tuplas para ejecutarlas. En este punto, se decide implementar los métodos para crear las reglas, utilizando el formato de mapas que utiliza el DAO de ETHiC, y posteriormente se implementan métodos para realizar una conversión entre el formato de la base de datos, y el utilizado por el motor de reglas. Durante la creación de las reglas, se realiza una validación de los campos introducidos, para evitar crear reglas mal formadas (ver figura 5.4).

Para finalizar la iteración, se integran unas reglas definidas por el equipo de Arce Clima utilizando los métodos implementados.

5.2 Frontend: Ampliación de la interfaz

Tras una interrupción en el desarrollo se reanuda este con la implementación de la interfaz web con la que utilizar el sistema creado anteriormente.

5.2.1 Actualización del sistema

Durante el parón del proyecto se actualizaron varias versiones de los frameworks utilizados en ETHiC. Dado que algunas versiones estaban ya obsoletas se realizó la actualización y corrección de errores ocasionados por los cambios.

5.2.2 Controlador

ETHiC utiliza Mochiweb [12] como servidor HTTP para crear una API Rest. Se utiliza un interceptor para delegar las distintas peticiones en varios controladores en función del módulo requerido. En el interceptor se definen que tipos de usuarios pueden acceder a las funcionalidades. Como ya se mencionó en la primera parte, se crea pues un controlador específico para las reglas. Este controlador delega en el modelo la funcionalidad, y devuelve la petición utilizando Mochiweb.

5.2.3 Vista

Esta capa está implementada en Angular, como una aplicación que utiliza la API Rest expuesta con Mochiweb. Angular utiliza Typescript para definir la lógica, HTML con unas etiquetas especiales de Angular para definir la interfaz y CSS3 para aplicar estilos. El markdown especial es del estilo *ngIf*, para mostrar condicionalmente elementos, o *ngFor* para generar múltiples elementos a partir de una lista.

```
check_fact({"equals", Var ,Expected_value}, Id) ->
    case get_data(Var, Id) of
        Expected_value -> true;
        _ -> false
    end;

check_fact({"not_equals", Var ,Expected_value}, Id) ->
    not check_fact({equals, Var, Expected_value}, Id);

check_fact({"less_than", Var ,Expected_value}, Id) ->
    case get_data(Var, Id) of
        Data -> Data < Expected_value
    end;

check_fact({"more_than", Var ,Expected_value}, Id) ->
    case get_data(Var, Id) of
        Data -> Data > Expected_value
    end.
```

Figura 5.3: Comprobación de hechos en las reglas

```
#{"name" := _Name, "operator" := Operator, "facts" := Facts,
    "actions" := Actions} = R = maps:with(?ATTR, Rule),
  Valid = valid(R),
  ValidOperator = valid_operator(Operator),
  ValidFacts = lists:map(fun valid_fact/1, Facts),
  ValidActions = lists:map(fun valid_action/1, Actions),
  Val = Valid ++ ValidOperator ++ ValidFacts ++ ValidActions,
  case ethic_validation:check(lists:flatten(Val)) of
  ...
```

Figura 5.4: Validación de una regla

El código se organiza en, componentes, servicios y modelos. Se crean dos componentes, que utilizan el mismo servicio para acceder a la API Rest. Un componente muestra la lista de reglas y el otro se utiliza para editar o crear reglas nuevas.

Los componentes en Angular se pueden utilizar dentro de otros componentes como etiquetas nuevas HTML, facilitando ser reutilizados. Los componentes (ver figura 5.5) tienen una parte en Typescript, otra, el template, en HTML y opcionalmente CSS3. La parte en HTML utiliza etiquetas especiales para comunicarse con el código Typescript. Tiene dos modelos, conexión en una única dirección, o en dos direcciones. La comunicación en una dirección solo se produce una única vez, aunque se actualicen los datos. Mientras que la comunicación en dos direcciones se encarga de que la vista siempre esté actualizada. En el ejemplo la sintaxis *rule.name* es una comunicación en una única dirección, ya que no se requiere que el título se actualice continuamente, mientras que *[(ngModel)]* sería comunicación en dos direcciones. Se observa como es especialmente útil para actualizar los datos de un formulario.

```
<navbar [installation]="installation"
    [(search)]="filterCollapsed"></navbar>
<div class="col-xs-12 col-sm-12 col-md-10 col-md-offset-2
    inner-wrapper">
  <div class="hidden-xs well container-fluid">
    <installation-info
    [installation]="installation"></installation-info>
  </div>
  <div *ngIf="rule" class="hidden-xs hidden-sm panel-heading
    container-fluid">
    <h4>{{ rule.name }}</h4>
  </div>
  <form
    [formGroup]="ruleForm"
    (ngSubmit)="submit(ruleForm.value)"
    class="form-group well"
  >
    <label for="name" translate>Name</label>
    <input
      type="text"
      id="name"
      formControlName="name"
      [(ngModel])="rule.name"
      [class.has-error]="!ruleForm.controls['name'].valid"
      class="form-control"
      required
      autofocus
    />

...
```

Figura 5.5: Ejemplo componente HTML

En Angular también se utiliza el patrón observador [13] , en el que un objeto llamado *sub-*

ject mantiene una lista de *observers* a los que notifica en caso de producirse cambios. Resulta de especial utilidad al hacer llamadas a la API, ya que se puede realizar distinta acciones en función de si el funcionamiento es correcto, o se produce un error (ver figura 5.6).

```
this._rulesService.createRule(this.rule).subscribe(
      (res) => {
        this._router.navigate([
          `/installations/${this.id}/rules`,
          { type: "success", message: "Rule Created" },
        ]);
      },
      (err) => {
        this._router.navigate([
          `/installations/${this.id}/rules`,
          { type: "error", message: "Error Rule Create" },
        ]);
      }
```

Figura 5.6: Ejemplo Observer

En proyectos Angular, toda la lógica común entre varios componentes, como conectarse a un servidor, se separa en servicios. En este caso se crea un servicio para comunicarse con los endpoints de la API Rest correspondientes al motor de reglas.

El diseño de la aplicación ya estaba definido, por lo que se utilizó el estilo existente para la interfaz (ver figuras 5.7 y 5.8).

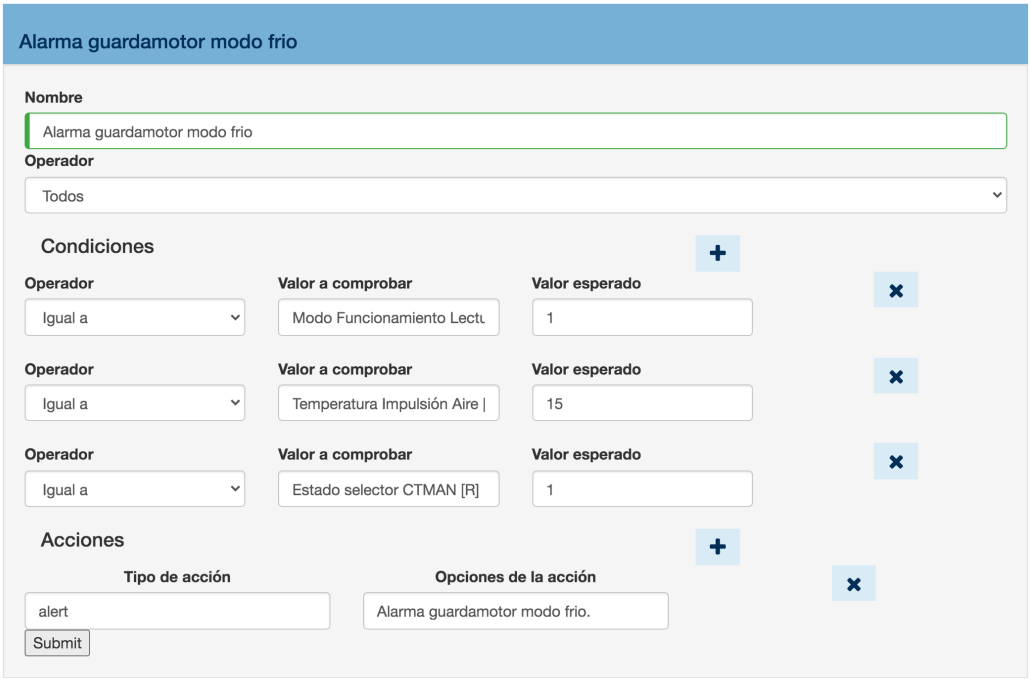

Figura 5.7: Edición regla

Figura 5.8: Listado de reglas

Capítulo 6
Validación

En este capítulo se describen las pruebas realizadas para el proyecto.

6.1 Pruebas de unidad

Una prueba unitaria comprueba el funcionamiento correcto de una unidad de código de manera aislada. En este caso se prueba la ejecución de diversas reglas sobre el motor de reglas para verificar su correcto funcionamiento. Se han creado diversos casos de prueba para comprobar todos los posibles tipos de hechos y acciones, comprobando que se ejecutan sólo al cumplirse las condiciones introducidas.

La aplicación ya contaba con una *suite* de pruebas, implementadas utilizando Common Test [14]. Common Test es un framework de Erlang que permite implementar pruebas automáticas de sistemas enteros.

En Common Test se organizan las pruebas en *groups* y *suites* en función de lo que vaya a probarse. Common Test permite que cada grupo, y cada suite, sean independiente de lo anterior facilitando el acceso a funciones para preparar la ejecución, y funciones para limpiar los datos creados después de la ejecución de cada suite, grupo o incluso tras la ejecución de cada caso. En la figura 6.1 se puede ver un ejemplo de las funciones *init_per_suite* y *end_per_suite* para preparar la ejecución y limpiar los datos generados tras esta. Al inicio de los tests se importa una instalación de prueba y se cargan las reglas, y al terminar se eliminan todos los datos creados al final la ejecución.

Para cada test que ejecuta Common Test este lanza un proceso dedicado en el que lanza la función del test, y en un proceso paralelo una función que mide el tiempo de ejecución. Si la función termina con una excepción, o tarda demasiado la ejecución el test falla.

Common Test permite además definir unas propiedades para definir como será la ejecución de los tests dentro de un grupo. Tiene opciones para ejecutar los tests aleatoriamente, en paralelo, secuencialmente o la opción para repetirlos. A su vez es posible que al terminar

cada test este pase una configuración al siguiente test.

Como otros frameworks de tests, Common Test genera un informe (ver figura 6.2) tras la ejecución con una interfaz en HTML. Este informe contiene una lista con todos los tests ejecutados y su resultado. Si falla el test permite acceder a el para ver más en detalle las razones del fallo.

6.2 Pruebas de integración

Las pruebas de integración verifican, tras realizar las pruebas unitarias, que el código añadido se integra con el resto del sistema. En este caso al realizar las pruebas unitarias se han utilizado varios módulos que utiliza el sistema fuera de los tests. Se ha utilizado el acceso al DAO con una base de datos de prueba, y se ha utilizado el sistema de alertas para la ejecución de las reglas.

6.3 Pruebas de aceptación

Las pruebas de aceptación se realizan en la fase final del desarrollo con el fin de determinar si la aplicación cumple los requisitos definidos por el usuario final. En el caso de este proyecto, la implementación del motor de reglas, junto a la implementación de las primeras reglas fue verificada por el equipo de Arce Clima, confirmando su correcto funcionamiento. Por otro lado, la interfaz web, con las funcionalidades para añadir, borrar y modificar reglas no pudieron ser verificadas por los usuarios finales.

6.4 Pruebas no funcionales

El objetivo de las pruebas no funcionales es verificar los aspectos del software no directamente relacionados con las funciones del sistema, como son el rendimiento, la fiabilidad o la escalabilidad. Durante el desarrollo se probó un diseño en el cual las reglas se ejecutaban en un sólo proceso. Este diseño demostró ser mas lento por la ejecución secuencial de todas las reglas. Cuantas más reglas, más tardaba en ejecutarse. Al ser independientes las reglas se decidió aprovechar la concurrencia de Erlang y ejecutar cada regla en un proceso independiente. Esto no solo garantiza la escalabilidad y el rendimiento de la aplicación, sino que trajo consigo la ventaja de que si una regla falla, no afecta al resto de reglas, añadiendo fiabilidad al sistema.

```
init_per_suite(Config) ->
    ethic_app:ensure_start(),
    Map = #{
    ...
    },
    #{"_id" := Id} = ethic_installations:create(Map),
    load_rules(),
    [Ws] = ethic_installation_sup:find_websocket_child(Id),
    [Event] = ethic_installation_sup:find_child(dynamic),
    [{id, Id}, {event, Event}, {ws, Ws} | Config].

%% @doc Clean up after test suite execution.
%% @end
end_per_suite(_Config) ->
    ethic_dao:delete_all("data"),
    ethic_dao:delete_all("rules"),
    ethic_dao:delete_all("installations"),
    application:stop(ethic),
    ok.
```

Figura 6.1: Ejemplo funciones de una suite

Num	Module	Group	Case	Log	Time	Result	Comment
	ethic_alert_SUITE		init_per_suite	< >	0.731s	Ok	
	common_test	alert_tests	init_per_group	< >	0.004s	Ok	start of alert_tests
1	ethic_alert_SUITE	alert_tests	find_all_error_test	< >	0.006s	Ok	
2	ethic_alert_SUITE	alert_tests	deactivate_test	< >	0.006s	Ok	
3	ethic_alert_SUITE	alert_tests	update_notfound_test	< >	0.001s	Ok	
4	ethic_alert_SUITE	alert_tests	find_filter_error_test	< >	0.002s	Ok	
5	ethic_alert_SUITE	alert_tests	find_error_test	< >	0.001s	Ok	
6	ethic_alert_SUITE	alert_tests	find_filter_test	< >	0.015s	Ok	
7	ethic_alert_SUITE	alert_tests	update_empty_test	< >	0.003s	Ok	
8	ethic_alert_SUITE	alert_tests	update_test	< >	0.005s	Ok	
9	ethic_alert_SUITE	alert_tests	find_test	< >	0.001s	Ok	
10	ethic_alert_SUITE	alert_tests	find_all_test	< >	0.001s	Ok	
11	ethic_alert_SUITE	alert_tests	create_empty_test	< >	0.001s	Ok	
12	ethic_alert_SUITE	alert_tests	find_all_empty_test	< >	0.004s	Ok	
13	ethic_alert_SUITE	alert_tests	deactivate_error_test	< >	0.001s	Ok	
14	ethic_alert_SUITE	alert_tests	create_test	< >	0.003s	Ok	
15	ethic_alert_SUITE	alert_tests	find_all_pag_test	< >	0.007s	Ok	
	common_test	alert_tests	end_per_group	< >	0.000s	Ok	end of alert_tests
	common_test	alert_tests	init_per_group	< >	0.000s	Ok	start of alert_tests

Figura 6.2: Ejemplo interfaz Common Test

Capítulo 7

Metodología

En esta sección de la memoria se discute la metodología aplicada en el proyecto.

7.1 Kanban y las metodologías ágiles

7.1.1 Metodologías ágiles y metodologías tradicionales

Las metodologías de software tradicionales hacen énfasis en realizar una planificación total de todo el proyecto, y una vez se realiza esta es cuando comienza el ciclo de desarrollo. Este tipo de metodologías tienden a no adaptarse fácilmente a los cambios, por lo que no son adecuadas para proyectos donde los requisitos no son fijos desde el principio o pueden variar durante el desarrollo del proyecto. Para paliar estos problemas, nacen las metodologías ágiles. Las metodologías ágiles adoptan ciclos de desarrollo más cortos, pequeños lanzamientos, diseños más simples y una refactorización continua. La principal diferencia entre las metodologías tradicionales y las ágiles, es la capacidad de adaptarse al cambio. En una metodología ágil, si se requiriese un cambio importante no se requeriría parar a todo el equipo para reiniciar el proceso, mientras que las metodologías tradicionales tienden a ”congelar” los requisitos. Algunas metodologías ágiles son Scrum, Lean o Kanban.

7.1.2 Kanban

Kanban es una metodología surgida en las fábricas de Toyota en los años 40 como sistema para implementar la fabricación ”just-in-time”. Comenzó como un sistema para gestionar el inventario a todos los niveles de la producción. Su implantación supuse un éxito para Toyota que terminó adoptando esta metodología en toda la compañía.

Con el tiempo otras compañías de distintas industrias adoptaron la metodología, siendo la más relevante la industria del Software.

7.1.3 Principios de Kanban

Kanban está basado en 3 simples principios.

- **Visualizar el flujo de trabajo.** Cuando se está trabajando en equipo se deben de tener muchas cosas en cuenta para colaborar eficientemente, cosas como quien está trabajando en que, que problemas están sin resolver. Se tiene que poder seguir el progreso del trabajo.

- **Limitar la capacidad de trabajo en curso.** Existe la necesidad de gestionar la cantidad de tareas que se están realizando. Si el equipo asume demasiadas tareas, estas tardarán en completarse y habrá gente que no tendrá tareas asignadas. Mientras que si la gente asume muy pocas tareas siempre habrá tareas que no están asignadas a nadie.

- **Gestionar el flujo de trabajo.** El flujo de trabajo no debe pararse, las tareas deben fluir continuamente. En el desarrollo de software existen diferentes tipos de problemas que ralentizan el desarrollo de un proyecto. Estos pueden ser tan sencillos como dejar trabajo a medias, cambiar continuamente entre tareas, esperar por una tarea pendiente o implementar funcionalidades extra no necesarias. Evitar estos problemas es esencial para el desarrollo.

7.1.4 Herramientas Kanban

La principal herramienta utilizada por el método Kanban es el tablero Kanban. Este tablero contiene múltiples columnas que representan las diferentes etapas por las que pasan las tareas. Generalmente en desarrollos de software estás tienden a ser del estilo: Pendiente, En Progreso, Hecho, en función del estado en que se encuentre la tarea. En desarrollos mas complejos se añaden columnas para las tareas pendientes, las tareas en pruebas o las que están en revisión.

Kanban significa en japonés ”tarjeta visual”. Estas tarjetas representan cada una de las tareas a realizar. Son una forma de visualizar, para todo el equipo, la información necesaria sobre una tarea. Suelen asignarse a miembros del equipo, lo cual facilita saber quien está trabajando en que de una manera sencilla y visual.

Las tarjetas sobre el tablero muestran donde se están produciendo atascos, cuellos de botella y donde se está avanzando más de una forma visual. Este tablero además no es necesario que sea privado. Se puede dar acceso a él al cliente o usuario final de la aplicación, que puede ofrecer una retroalimentación sobre las tareas, su ordenación o incluso crear nuevas tareas. Al ser Kanban una metodología ágil soporta perfectamente este tipo de cambios, y al realizarse entregas en plazos muy cortos no deberían suponer problemas a la hora del desarrollo.

7.1.5 Métricas en Kanban

Como en la metodología Kanban los requisitos van surgiendo y modificándose a lo largo del desarrollo, resulta difícil realizar estimaciones en cuanto a la duración del proyecto. Pese a esto existen diversas métricas que se pueden utilizar como son el ”Lead Time”, que mide el tiempo que transcurre entre el momento en el cual se pide una tarea (creación de la tarjeta) hasta su finalización. O el ”Cycle Time” que mide el tiempo que la tarea está en la columna ”En proceso”, es decir, el tiempo que se está trabajando activamente en la tarea.

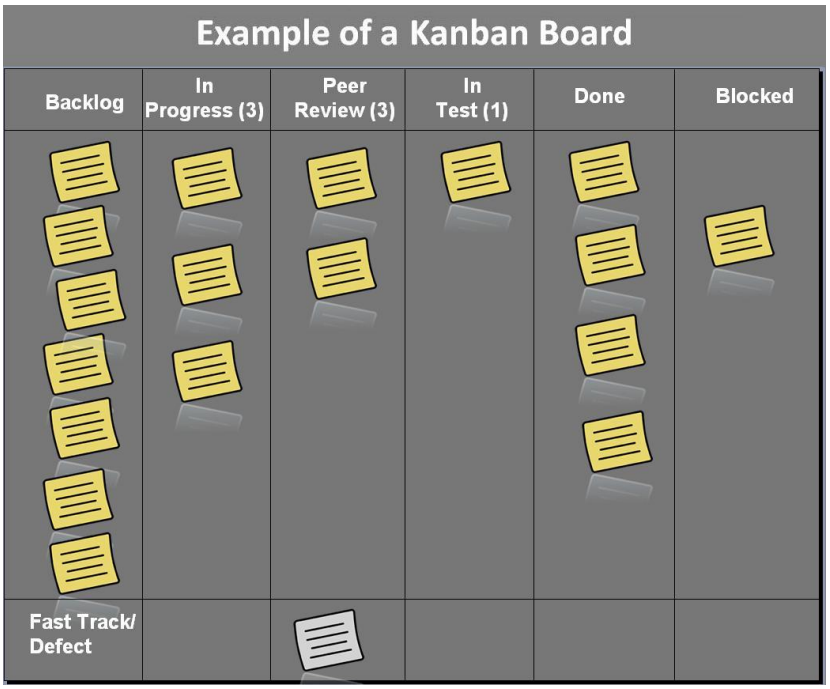

Figura 7.1: Ejemplo de tablero Kanban

7.2 Kanban en el proyecto

Pese a que en Kanban no se contemplan las iteraciones, como ya se comentó en el capítulo 5 , en este proyecto hay dos fases separadas en el tiempo que trataremos como iteraciones. En ambas iteraciones se utiliza un tablero Kanban con las siguientes columnas:

- **TODO:** En esta columna se colocan las funcionalidades que se van a desarrollar. Esta columna no tiene límite de tareas.
- **DOING:** En esta columna se colocan las tareas en las que se está trabajando actualmente. El límite es una por desarrollador.

- **TEST:** Cuando se termina una tarea pero deben de ser evaluadas.
- **PEER REVIEW:** En esta columna se colocan las tareas para que las revise otro desarrollador del equipo. Generalmente se hacía haciendo una pull request en el repositorio del proyecto.
- **DONE:** Aquí irían las tareas terminadas.
- **BLOCKED:** En esta columan se sitúan las tareas que no pueden realizarse hasta solucionar algún problema.

Esta estructura de columnas se mantuvo durante las dos partes del proyecto. En la primera parte se produjeron algunas reuniones con técnicos de Arce Clima para verificar funcionalidades, y estos tenían también acceso al tablero. Pese a esto la comunicación entre el equipo de desarrollo y el equipo de Arce Clima no fue muy fluida.

Durante la segunda mitad, cuando se implementó la interfaz web, ya no se contaba con esta revisión por parte del cliente, por lo que no recibió una validación del equipo de Arce Clima.

7.3 Estimación de esfuerzo

A continuación se detalla el esfuerzo dedicado en horas/persona (h/p) a varios conjuntos de tareas.

- **Familiarización con ETHiC:** Al principio del desarrollo se realizaron varias tareas para familiarizarse con el sistema. Se emplearon 40 h/p a estas tareas.
- **Obtención de información sobre sistemas de reglas:** Se emplearon 60 h/p en obtener información sobre sistemas de reglas existentes.
- **Reuniones con el equipo de Arce Clima:** Se emplearon 8 h/p a las reuniones con el equipo de Arce Clima.
- **Desarrollo motor de inferencia:** En el desarrollo del motor de inferencia se emplearon que necesitó 100 h/p.
- **Pruebas reglas:** En el desarrollo de los tests en Common Test se emplearon 20 h/p.
- **Actualización de la aplicación:** Tras el parón en el desarrollo fue necesario actualizar varios frameworks utilizados en la aplicación. Para esto se emplearon 8 horas/persona.
- **Análisis Angular:** Para familiarizarse con la estructura de las aplicaciones en Angular se emplearon 20 horas/persona.

- **Integración Angular:** En implementar los componentes en Angular se emplearon 40 h/p.
- **Memoria Proyecto:** En la memoria del proyecto se emplearon 40 h/p.

Total de esfuerzo: 336 horas/persona Por tanto, teniendo en cuenta un coste de 40€/hora el coste total del proyecto serían 13440€.

Capítulo 8
Conclusiones

En este capítulo de la memoria se exponen las conclusiones del proyecto y el posible trabajo futuro.

8.1 Objetivos alcanzados

Este proyecto tenía dos objetivos principales: primero el desarrollo de un motor de reglas que permitiese automatizar tareas dentro del sistema ETHiC, y por otro lado una interfaz web con la que gestionar las tareas básicas de gestión del motor de reglas. Ambos han sido logrados satisfactoriamente.

8.2 Lecciones aprendidas

A lo largo del proyecto se hizo uso principalmente de Erlang como lenguaje para el desarrollo del *backend* y de Typescript para el desarrollo del *frontend*. Erlang/OTP con sus *behaviors* facilitó la fase de diseño. La arquitectura líder/trabajador facilita la integración del sistema de reglas y mantener un buen rendimiento pese a la ejecución de un elevado número de reglas. La separación en capas independientes facilita la incorporación del sistema y la modificación de este.

Por otro lado, Erlang resultó problemático durante la actualización del sistema a las últimas versiones. Una de las carencias que tiene el lenguaje son sus mensajes de error poco claros que dificultaron localizar donde era necesario realizar cambios.

Respecto a la web, Angular con su arquitectura basada en componentes y servicios facilitó integrar una interfaz para visualizar y crear las reglas. El sistema de reglas permite ejecutar el tipo de reglas solicitado por el equipo de Arce Clima, pero podrían haberse delegado más funcionalidades en él para aprovechar sus capacidades.

Es importante destacar lo aprendido durante el desarrollo. ETHiC es un proyecto con

cierta madurez, que utiliza diversas tecnologías en las que se ha ganado experiencia, como son Angular y Erlang/OTP. Se ha implementado un sistema inteligente basado en reglas, uno de los sistemas tratados en la carrera, en una aplicación con usuarios finales. Pese a ser uno de los primeros sistemas inteligentes que se crearon, permite cumplir los requisitos pedidos por el equipo técnico, y es posible extenderlo con más capacidades. El formato de las reglas facilita que los usuarios técnicos modifiquen las reglas existentes, y añadan reglas nuevas de ser necesario. Durante el desarrollo, se han aplicado metodologías ágiles y se ha ganado experiencia en la extracción de requisitos durante las pocas reuniones que se pudieron realizar. Y se ha ganado experiencia para comenzar a trabajar en un proyecto con una arquitectura ya definida, y a realizar cambios en él manteniendo sus funcionalidades intactas.

8.3 Trabajo futuro

A pesar de cumplirse los objetivos marcados por el cliente con el sistema de reglas que necesitaban, y la interfaz para poder gestionarlo, existen múltiples posibilidades de expansión, entre las que podemos mencionar:

- **Utilizar datos históricos:** actualmente las reglas se aplican sobre los datos cada vez que el sistema los obtiene. Dotar al sistema de capacidades para utilizar datos históricos permitiría al sistema reaccionar antes ante posibles fallos.
- **Dotar al sistema de más acciones:** al tratarse de sistemas industriales no es recomendable dejar que el sistema experto realice acciones por si mismo sobre los actuadores. Pero si sería posible que el sistema genere recomendaciones de que acciones recomendaría tomar.
- **Ejecución más inteligente de las reglas:** Actualmente se ejecutan todas las reglas del sistema, si se pudiese extraer características comunes entre estas con un método automático podría simplificarse la ejecución lanzando procesos de una forma más optimizada.

Bibliografía

[1] L. M. Castro, J. D. Fernández, and C. L. Pampín, "Making everybody comfortable with erlang: A scada system for thermal control," in *Proceedings of the 15th International Workshop on Erlang*, ser. Erlang 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 56–57. [En línea]. Disponible en: https://doi.org/10.1145/2975969.2975976

[2] S. J. Russell and P. Norvig, *Artificial Intelligence A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[3] A. A. Grosan C., *Rule-Based Expert Systems. In: Intelligent Systems. Intelligent Systems Reference Library*, 1st ed. Springer, 2011.

[4] J. Armstrong, M. Williams, and R. Virding, *Concurrent Programming in Erlang*, 2nd ed. Prentice Hall, 2014.

[5] F. Hébert, "Learn you some erlang for great good!" 2013. [En línea]. Disponible en: https://learnyousomeerlang.com/

[6] "Angular framework." [En línea]. Disponible en: https://angular.io/

[7] S. Chacon and B. Straub, *Pro Git*, 2nd ed. Apress, 1993.

[8] D. Anderson, *Kanban - Successful Evolutionary Change for your Technology Business*, 1st ed. Blue Hole Press, 2010.

[9] "2.4 designing rule bases." [En línea]. Disponible en: http://www.billbreitmayer.com/rule_based_systems/rule_based_design.html

[10] "Seresye." [En línea]. Disponible en: https://github.com/afiniate/seresye

[11] "Zippers." [En línea]. Disponible en: https://ferd.ca/yet-another-article-on-zippers.html

[12] "Mochiweb." [En línea]. Disponible en: https://github.com/mochi/mochiweb

[13] "W3design observer." [En línea]. Disponible en: http://w3sdesign.com/?gr=b07&ugr=struct

[14] "Commontest." [En línea]. Disponible en: http://erlang.org/doc/apps/commontest/basicschapter.html.